Отчет по лабораторной работе №1

> Валеев Нурсан M3437

Описание задачи конкурса GDCC (Test 5)

Нужно соптимизировать Finite State Machine (FSM) для RangeCoder из условия. RangeCoder хранит контексты длины 16-23 бит и каждый контекст проходит по FSM независимо от других.

FSM - это направленный граф, где из каждой вершины выходит по 2 ребра (по нулю и по единице) и у каждой вершины есть вероятность прохода по нулю.

Далее RangeCoder проходит по каждому биту в файле и для каждого контекста (который набран на этот момент) делает переход по соотвествующему ребру в FSM, умножая вероятность на текущее число (аналогично арифметическому кодированию).

В ниже следующих подходах оптимизируется конкретная FSM (уже сгенерированная) под конкретный файл.

Подход #1

Можно раздать более “правильные” вероятности на вершинах FSM, для этого нужно пройтись по файлу, не кодируя его, посчитать, сколько переходов по 0 и 1 было сделано и тогда конечная вероятность будет равна (для каждой вершины):

```
p = count_0 / (count_0 + count_1)
```

Такой подход используется в FSM0a.txt, но он подсчитывается для конкретного файла, поэтому скорее всего сжатие FSM0.txt на другом файле будет работать лучше, чем FSM0a.txt подсчитанное на другом файле.

По таблице результатов GDCC видно, что FSM0a “проигрывает” FSM0 на test1_demo и test1, так как подсчитан для book1.

Итог: можно генерировать граф из вершин, а вероятности раздавать после прохода по файлу.

Подход #2

Заметим, что если у вершины есть несколько “родителей”, то можно отцепить одного из родителей, сгенерировав ему новый граф, эквивалентный предыдущему и пусть новое ребро в этот граф. Тогда влияние этого родителя на эту вершину уменьшится и его вероятности “плавнее” распределяться в новом графе (который будет несвязен с предыдущим, так как в него можно только перейти, а вернуться нельзя).

Такая эвристика увеличивает количество вершин минимум в 2 раза, при этом ограничение конкурса - 32768 вершин.

При этом, появляются вершины, которые ни разу не были посещены за обход файла. Такие вершины удаляются, а остальные вершины FSM перенумеровываются, поэтому алгоритм не заканчивается за **O(log (32768 / start_size))**.

Подход #3

После применения подхода #2, я заметил, что на некоторых вершинах появляются вероятности 0 и 32767 (вероятность записывается с точностью 15 бит). Т.е. это безусловные переходы по 0 и 1 битам, которые не увеличивают размер конечного файла. Иногда такие цепочки заканчиваются циклами.

Можно объединять такие цепочки, если у них есть общий суффикс, например:

1. Цепочка длины 10: 0101010101
2. Цепочка длины 5: 01101

Тогда, можно найти все цепочки определенной длины, и каждую меньшую цепочку “склеивать” с цепочкой большого размера, но с суффиксом равным меньшей цепочке.

Это оптимизация вершин графа, но compression rate при этом уменьшается.

Конечное решение

Подход #3 в совокупности с подходом #2 могут бесконечно расширять и сжимать граф, находя все лучшие и лучшие FSM. Но если применять только подход #2, то сходимость алгоритма гораздо быстрее (подход #3 может сильно увеличивать конечный сжатый файл). Поэтому было выбрано использование подхода #2 (при этом, с подходом #1, так как он подразумевается в условии).

Дальнейшие улучшения

В следующей секции показаны результаты работы для FSM0 - bruteforce для book1bwt из 256 вершин. Она показывает лучшие результаты на оригинальных файлах и имеет наименьшее количество вершин. При этом, как дальнейшее улучшение - можно сгенерировать такой же файл (за большое процессорное время), по аналогии с FSM1-4 и сжать его до, например, 256 вершин брутфорсом (выбирать все попарные вершины и пытаться сжимать граф, выбирая наилучшее изменение конечного размера файла). Каждый шаг такого алгоритма будет работать за **O(N^2)**, но при этом, наибольшее N = 32768.

Результаты работы

FSM0 - оригинальный файл FSM0.txt
FSM0 R - реализация подхода #1
FSM0 E- реализация подходов #1 и #2 (итоговое решение)

Сравнение улучшений сжатия

Файл	Размер	FSM0	FSM0 R	FSM0 E
bib	111261	41108	39279	35300
book1	768771	295651	289113	281319
book2	610856	223320	217669	211721
geo	102400	74660	61067	56688
news	377109	162992	155363	150213
obj1	21504	11937	11332	10555
obj2	246814	94132	88819	85340
paper1	53161	21620	20502	17182
paper2	82199	32165	30915	27279
pic	513216	61587	57562	53309
progc	39611	15922	14872	11710
progl	71646	21977	20503	17302

Файл	Размер	FSM0	FSM0 R	FSM0 E
progp	49379	15157	14241	11028
trans	93695	29078	27384	23690

Итоговая таблица для **FSM0 E**

Файл	H(X)	H(X|X)	H(X|XX)	Затраты на символ	Размер сжатого файла
bib	5.20068	3.36410	2.30746	2.5382	35300
book1	4.52715	3.58451	2.81407	2.9275	281319
book2	4.79263	3.74521	2.73566	2.7728	211721
geo	5.64638	4.26428	3.45777	4.4288	56688
news	5.18963	4.09188	2.92274	3.1866	150213
obj1	5.94817	3.46414	1.40094	3.9267	10555
obj2	6.26038	3.87036	2.26542	2.7661	85340
paper1	4.98298	3.64602	2.33168	2.5857	17182
paper2	4.60143	3.52231	2.51358	2.6549	27279
pic	1.21018	0.82365	0.70519	0.8310	53309
progc	5.19902	3.60330	2.13389	2.3650	11710
progl	4.77009	3.21156	2.04350	1.9320	17302
progp	4.86877	3.18748	1.75505	1.7867	11028
trans	5.53278	3.35490	1.93059	2.0227	23690

Суммарный размер сжатых файлов - 992636 байт.

Приложение

Программа **gen.exe** - генерация fsm.txt по input файлу

```
gen.exe input fsm.txt
```

Программа **entropy.exe** - подсчет энтропии

```
entropy.exe file # fsm.txt from fsm/* for coressponding file
```

Программа **coder.exe** - неизмененный rangecoder из задания

```
coder.exe c input output fsm.txt # to compress
coder.exe d input output fsm.txt # to decompress
```

Исходные файлы **gen.cpp** и **coder.cpp**
Сгенерированные **fsm.txt** для всех файлов из задания (либо можно сгенерировать новые)